AN EFFICIENT OUTPUT-SENSITIVE HIDDEN-SURFACE REMOVAL ALGORITHM AND ITS PARALLELIZATION ‡

(Preliminary Version)

John H. Reif and Sandeep Sen

Computer Science Dept
Duke University,
Durham, N.C. 27706

ABSTRACT

In this paper we present an algorithm for hidden surface removal for a class of polyhedral surfaces which have a property that they can be ordered relatively quickly like the terrain maps. A distinguishing feature of this algorithm is that its running time is sensitive to the actual size of the visible image rather than the total number of intersections in the image plane which can be much larger than the visible image. The time complexity of this algorithm is O((k +n)lognloglogn) where n and k are respectively the input and the output sizes. Thus, in a significant number of situations this will be faster than the worst case optimal algorithms which have running time $\Omega(n^2)$ *irrespective of the output size (where as the output size k is* $O(n^2)$ *only in the worst case). We also present a parallel algorithm based on a similar approach which runs in time* $O(\log^4(n+k))$ *using O((n + k)/log(n+k)) processors in a CREW PRAM model. All our bounds are obtained using ammortized analysis.*

I. Introduction

The hidden-surface elimination problem (see [17] for an early history) has been a fundamental problem in computer graphics and can be stated in the following manner - given n polyhedral faces in a three dimensional environment and a projection plane, we wish to determine which portions of the polygonal boundaries (regions) are visible when viewed in a direction perpendicular to the projection plane. We are interested in a *object-space* solution (independent of the display device) for this problem. It has been shown that the worst case output size for hidden-surface elimination can be $\Omega(n^2)$for n segments and hence it is clear that the worst case optimal algorithms for these problems will have a running time of $\Omega(n^2)$. Recently McKenna[7] proposed an algorithm for the general problem which runs in $O(n^2)$ and hence is worst-case optimal.

A slightly different version is the hidden-line elimination problem, where we are concerned only with the visibility of the edges (and not regions). The algorithms for hidden-surface removal can be easily modified for the hidden-line elimination case but not vice-versa. There are algorithms for hidden line elimination in literature whose running time is sensitive to the intersections (of the projection of the segments) in the image plane, typically of the order of O(n+k)logn (for example see Nurmi[13] and Schmitt[14]). Very recently this was improved to O(nlogn + k + t) by Goodrich[15] where t is number of intersecting polygons on the image plane. However, in practice, the size of a displayed image can be far less than the number of intersections in the image plane. By size, we mean the number of edges and vertices of the displayed image as a (planar) graph. graph This happens because a large number of these intersections are occluded by visible surface and hence do not increase the complexity of the image. Our objective is to design an algorithm whose running time is sensitive to the final displayed image rather than the number of intersections. In this paper we design output-sensitive algorithms for a restricted class of surfaces like terrains which will hopefully give us a good grip on the more general problem.

The terrain maps are polyhedrons in 3-space (see Figure 0) which can be represented as

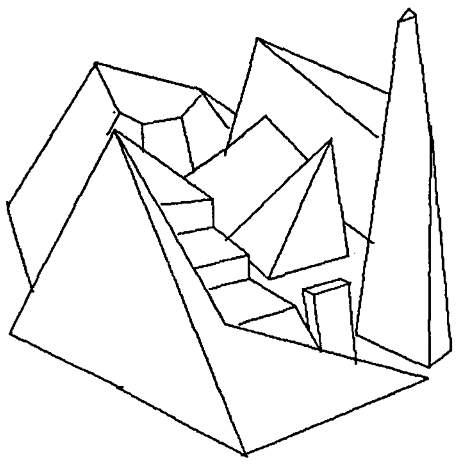

Figure 0: A typical scene as a terrain map.

a function of two-variables, for example z = f(x,y) wlog. Most geographical features can be represented in this manner. Another characteristic of these surfaces is that, the projections on the z-x and the z-y are monotone w.r.t. x and y axes(respectively). In fact this turns out to be a very useful property for making the algorithms somewhat simpler than any general Hidden-surface removal algorithm. In spite of that, it has been shown that the size of the visible image can be $O(n^2)$ in the worst case which is the number of intersections in the projection of the line segments into the image plane.

A commonly used technique is to process the surfaces in increasing distance from the viewing plane so that each point needs to be tested only once for visibility i.e. a point once pronounced as visible is not going to be altered later in the course of the algorithm (the same holds true for the occluded points). The origins of this approach can be found in [8]. However instead of performing the visibility test for each point on the display device so that the complexity of the algorithm is also dependent on the resolution of the display device, we do it in a device-independent manner. The output of our algorithm is a graph of the final image and not a pixel by pixel description of the image. Our techniques apply to terrain maps, whose edges can be ordered from 'front to back' very quickly. We conjecture that they can be extended to surfaces like star-shaped polyhedra without much difficulty.

‡ This research was supported by National Science Foundation Grant DCR-333-0846, by a grant from the Office of Naval Research under contract ONR-383-3246 and by Air Force contract AFSOR-313-5016.

We assume that the vertices of the terrain map are available as 3-tuples (x,y,z) of coordinates after the necessary transformations have been carried out. During the course of our algorithm, we maintain an upper profile of the segments processed upto any given point and test for the visibility of the current segment by intersecting it with this profile. The portion of the segment inside the upper profile (which is a simple monotone polygon) is not visible and hence is discarded. The upper profile may have to be updated with the portions of the segment that is visible. Thus the main procedure in our algorithm centers around detecting the intersection(s) of a line segment with a simple (actually a monotone) polygon. For this purpose, we use an efficient algorithm given by Chazelle and Guibas[1]. However, we need to modify their algorithm to suit a dynamic environment since the polygon (upper profile) is getting modified over the course of execution. In the next section, we shall review their algorithm in the context of making it dynamic. Cole and Sharir[3] have recently presented algorithms for fast processing of query rays emanating from a point above the terrain map. Though the overall approaches are somewhat similar, our work is more closely related to the problem of producing a graph of the displayed image without relating to the device-coordinates (similar to [7]). It is not clear how the algorithm in Cole & Sharir[3] can be modified to handle the hidden-surface elimination problem by making an object space version of it. In the following section we shall present the sequential algorithm and its analysis.

Next, we present a parallel algorithm which runs in polylogarithmic time using a number of processors dependent on the output size. A straight-forward parallelization of the sequential algorithm is not processor efficient. The constraint on the number of processors used requires us to take a non-conventional approach and use dynamic parallel data-structures.

We conclude with discussions on possible improvements and some open problems.

II. Intersecting segments with simple polygons

Given a simple polygon of n segments, how does one detect all the intersections with a query segment efficiently ? Chazelle and Guibas[3] provided a near optimal solution to this problem using an application of geometric duality. Their result can be summed up in the following manner:

Fact 1 : There exists an O(n) space data structure representing a simple polygon P which can be computed in time O(nlogn) which, when given a segment s intersecting P in k places, allows us to find these intersections in $O((k+1)\log(\frac{n}{(k+1)})$ time.

Though this is a near optimal (in the sense of an O(nlogn + k) time and linear space) algorithm, it involves a very complex data organization which does not appear to be suited for a dynamic environment i.e., one in which we may have to update this data structure periodically to accommodate a change in the polygon itself. For expository reasons we shall use a somewhat weaker result by sacrificing a factor of O(logn) in time and space complexity and then improve it. Before we describe our dynamic structure, we shall review their simplified algorithm in some detail.

Given a simple polygon, we construct a binary tree structure where each node represents a portion of the polygon and the leaves correspond to the triangles of the triangulated polygon. The size of the polygons associated with each node decreases geometrically with depth so that the tree has a logarithmic depth. At any level of the tree, the polygons associated with the nodes of the tree are disjoint (except for a shared edge which is a diagonal in the original polygon) and their union is the given polygon. The polygon is divided using Chazelle's[6] polygon cutting theorem which can be stated as follows :

Fact 2 : Let P be a simple polygon with N vertices $v_1,v_2,..v_N$ sorted along some axis. Then it is possible to find, in O(N) time, a pair of vertices v_i,v_j such that the segment v_iv_j lies entirely inside the polygon and partitions it into two simple polygons satisfying $C(P_1) \leq C(P_2) \leq 2C(P)/3$ where C(P) is the size of the polygon.
The following corollary is almost immediate

Corollary 1 : The binary tree corresponding to the polygon can be constructed in O(nlogn) time.

To detect the intersections, we use divide and conquer on the polygon using this tree. We need to review a few results from geometric duality for detecting intersections (Chazelle and Guibas[1]).

Fact 3 : Given a simple polygon, all the lines passing through a fixed side of the polygon form convex subdivisions in the 2 SP‡ with respect to the edges of the polygon it intersects first (see Figure 1).

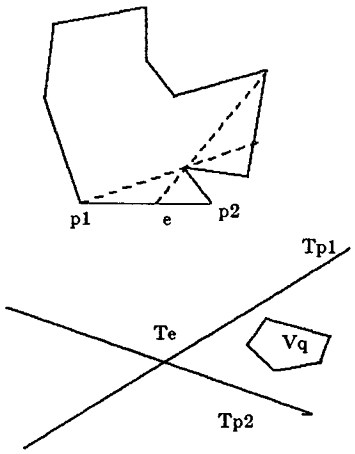

Figure 1 T_{p1} and T_{p2} are the duals of the points p1 and p2 T_e is the dual of the edge e. The dotted region is the dual of all rays intersecting the edge e and V_q is the convex region in the dual plane containing the duals of all the lines passing through e and q without intersecting the polygon in between.

These convex subdivisions are referred to as the visibility polygons and $P_{a,b}$ is the visibility polygon associated with a line which intersects sides a and b without intersecting any part of the polygon in between. The following ifact provides a bound on the total size of all the visibility polygons:

Fact 4 : Since each edge of the convex subdivisions (in the 2 SP) separates exactly two regions connected with two edges of the polygon, the total size of all the visibility polygons is at most twice the given polygon.

Remark : In future we shall use this fact as the size of the visibility polygons being O(P(n)) where P(n) is a simple polygon with n vertices.

The algorithm for computing the visibility polygons is based on divide and conquer, where the polygon is divided into two almost equal parts using fact 1. The visibility structures of each of these polygons is computed recursively and then they are merged in linear time leading to the following result :

Fact 5: [1] It is possible to compute the visibility polygons (with respect to a given side) of a simple polygon in O(nlogn) time and O(n) space, where n is related to the polygon size.

Informally, the line intersection algorithm can be viewed as following. From each node of the binary tree, we try to find the furthest node such that the line joining the diagonals represented by the nodes remains inside the polygon. Since each node is representative of a portion of the polygon and the 'cutting diagonal' of the two

‡ Two-sided plane or the dual plane, see [9] for details.

polygons of its children nodes, we look for a node of least depth in its subtrees such that the line does not intersect the polygon in between. In other words, from a node v, we try to find a visibility polygon $P_{v,x}$ such that the dual of the line lies inside $P_{v,x}$ and x is a diagonal associated with a node which has the least depth among all eligible nodes (see Figure 2). This gives rise to a data structure, which is a binary tree (of the divided polygon), where each node is augmented by pointers to the rightmost (leftmost) nodes at each level of its left (resp. right) subtrees. Notice that each of these pointers corresponds to a visibility polygon $P_{v,x}$ where v and x are the diagonals associated with the nodes. The size of the tree (augmented with the extra pointers) is still O(n); however the size of the visibility polygons associated with each level is O(n) thus amounting to a total space of O(nlogn). The algorithm for detecting the intersection of a line with the polygon is now quite simple. From a point on the line inside the polygon, we hop to a node such that the line does not intersect the polygon and repeat the procedure from the new node until we reach a leaf from where we can detect the intersection in constant time (since it is a triangle). Each such "hop" from one node to another involves a point location in a convex polygon (in the dual plane), and the total number of such searches is bounded by the height of the tree i.e. O(logn).

Fact 6: It is possible to test containment of a point in a convex polygon in O(logn) time given a preprocessing time of O(n).

The preprocessing can be absorbed in the preprocessing cost for building the data structure and thus we can state the following result of Chazelle and Guibas[1]

Lemma 1 : Given a simple polygon, there exists a O(nlogn) space data structure which can be constructed in O(nlogn) time which allows us to detect the intersections between a line segment and the polygon in $O(\log^2 n)$ time per intersection.

Given the above framework, we shall now extend it to a dynamic environment, where not only do we detect the intersections but also modify the polygon by joining the intersections with a straight line (see figure). The problem is primarily two-fold :

(1) We need to modify the data structure, specifically the visibility polygons very fast with the introduction of the new segments.

(2) We have to keep the underlying tree balanced, so that the depth of the tree remains logarithmic in number of leaves.

An eligible candidate for the underlying balanced tree is a class of weight balanced trees BB(α) tree (Mehlhorn[4]). Appendix 1 gives a description of the general properties of these weight-balanced trees. We outline here some of the more important characteristics of this tree relevant to our needs:

(i) These trees have logarithmic height in the number of nodes.

(ii) The amortized cost for m deletions or insertions is O(m) rotations, and the number of rotations geometrically decrease as we get closer to the root.

As it turns out both of these properties lead us to an efficient algorithm. We shall now describe the elementary operations on this data structure in the realm of our algorithm viz. insertion, deletion and rotation.

Insertion

In a way insertion and deletion are quite related - the insertion of a segment may lead to deletion of one or more segments (see Figure 3). By insertion of a line segment, we have to modify a number of visibility polygons on its path from the root to the leaves.

Claim 1: The number of visibility polygons that have to be modified by insertion of a line segment is O(logk) where k is the number of nodes in the tree.

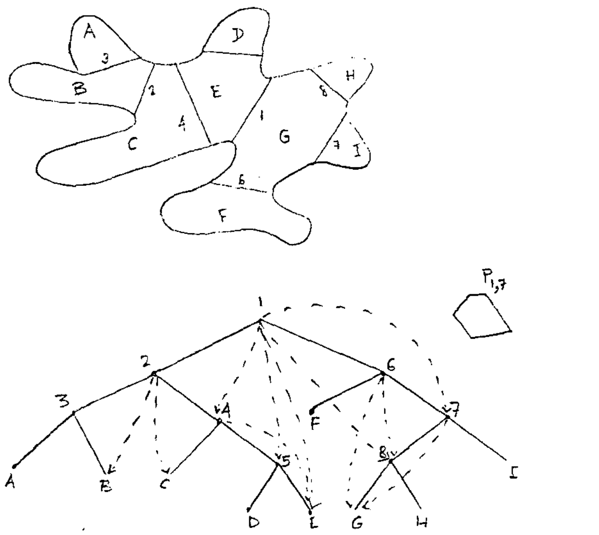

Figure 2 Decomposition of a polygon using the polygon cutting theorem. The dotted lines represent the visibility polygons enclosing the dual of the line segments passing through the portion of the polygon between the diagonals (correponding to the nodes) without intersecting the polygon in between.

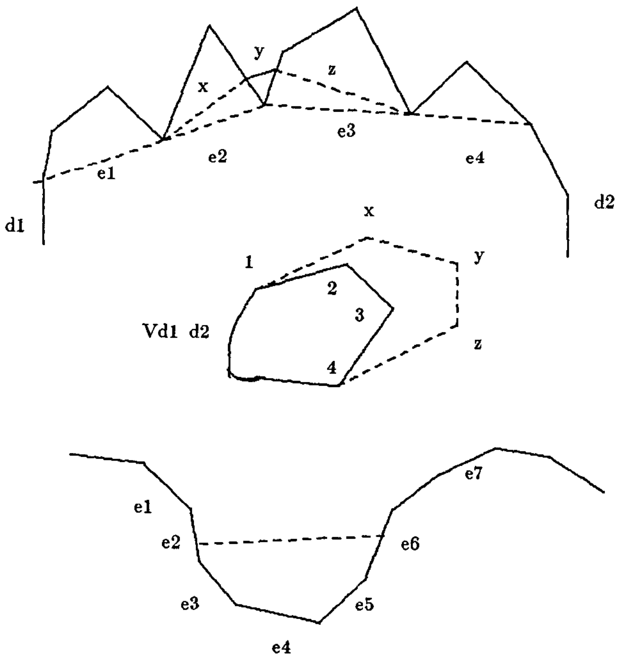

Figure 3: Updating the visibility polygons due to a new intersection. 'x', 'y' and 'z' are the new vertices in the polygon Vd1d2.

proof : Consider a case where the new segment passes over a single diagonal, which leads to the deletion of the corresponding node and insertion of two nodes corresponding to the two end points of the new segment (see Figure 3a) This affects all the visibility polygons $P_{x,y}$ such that the deleted diagonal lies between x and y. There can be at most one such polygon in each level of the tree (since the visibility polygons are non-overlapping). If the new segment deletes more than one diagonal (see Figure 3b), we can do it in time proportional to the number of diagonals by viewing the process as introduction of a sequence of hypothetical segments each of which deletes a diagonal such that the resultant picture looks the same. By repeating the procedure for deleting a single diagonal, we can charge the work done to the output size (since all the eliminated vertices corresponding to the diagonals are a part of the displayed image).

Claim 2: The modification in the polygon can be done by introducing each of the three points which are the duals of the new segments introduced.

proof: The vertices of a visibility polygon are the duals of the boundary edges of the polygon G (see Figure 3). The introduction of a new segment introduces at most 3 such new edges or equivalently 3 new vertices in a the dual polygon. Notice that the insertion of the new vertices can lead to the deletion of some existing vertices, i.e. the modified polygon is the convex hull of the vertices.

Fact 7:[10] The supporting lines can be found at a cost of O(log k) time where k is the size of the convex polygon.

Fact 8 [9] : The intersection of two polygons (also a convex polygon) can be found in O(m + n) time where the polygons have m and n vertices respectively.

Claim 3: A rebalancing operation (i.e. a rotation or a double rotation) can be carried out in time O(th(v)) where th(v) is the number of nodes in the subtree rooted at v (v is the vertex where rebalancing operation is being applied).

proof: We shall prove it for a single rotation - the proof for double rotation is similar (applying it twice). Let us denote the left subtree of v as a and the subtrees of w as b and c. Figure 4a shows a single rotation.

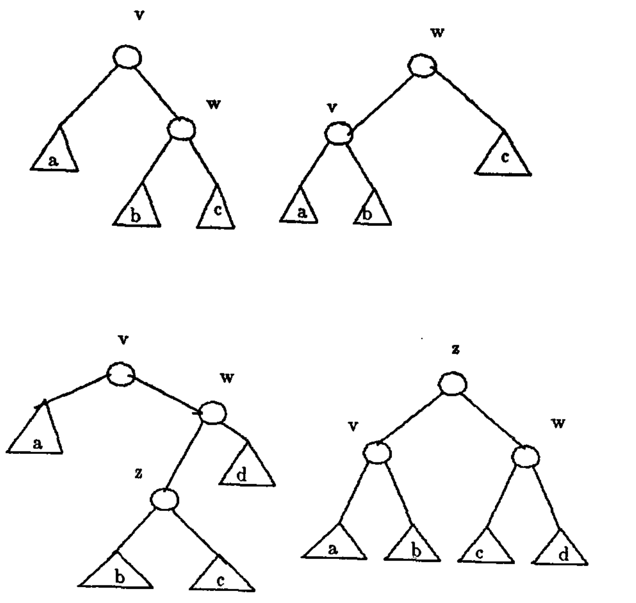

Figure 4: Rotation and double rotation in a BB(α) tree.

This may have the following effects on the visibility polygons.
(a) We have to recompute the visibility polygon $P_{parent(v),w}$.
(b) If there were a pointer from some ancestor of v to v (corresponding to some visibility polygon), we simply remove that pointer.

Actually there may be some more variants of the above two cases which can be handled very similarly. An important point to note is that the visibility polygons corresponding to a, b and c are unaffected. Analyzing the effects of (1) and (2) more carefully, we have to compute intersection of two convex polygons, where the size of each polygon is bounded by O(th(v)). From Fact 8, this can be computed in the same asymptotic time bounds. □

Fact 9: Let $1/4 \leq \alpha < 1-\sqrt{2}/2$ and let f be a non-decreasing function, then the total amortized rebalancing cost of m insertions and deletions can be bounded by $O(m \sum_{i=0}^{e\log n} f((1-\alpha)^{-i})(1-\alpha)^{i})$ where $e = 1/\log(1/1-\alpha)$

Remark : if $f(k) = O(m(\log^k m))$ then the total cost $= O(m \log^{(k+1)} m)$.

Lemma 2 : The total rebalancing cost corresponding to updates of the upper profile is O((k + n)logn) where k is the number of insertions and n is the input size.
proof : Follows from claim 3 and Fact 9 (use k = 0 in the previous remark).

III. Description of the algorithm and its analysis

Given the background of the previous section, we are almost ready to describe the main algorithm. The only detail left to be worked out is the computation of an ordering of the surfaces to be painted from front to back. For this we use the following scheme. We project the edges on the X-Y plane and now the ordering of the surfaces corresponds to ordering the edges on the plane in increasing distances of x. Fortunately, there is a closely related problem which can be used to do this.

Definition: A Chain $C = (u_1,u_2,..u_p)$ is a PSLG with vertex set $\{u_1,..u_p\}$ and edge set $\{(u_i,u_{i+1})\}$ where i = 1,2 ..p-1. A Chain is called monotone with respect to a straight line l if a line orthogonal to l intersects C in exactly one point.

Thus given the PSLG (which is the projection on the X-Y plane), we have to decompose it into a set of monotone chains with respect to the x axis. Note that this gives us a total ordering for the set of edges (see Figure 5).

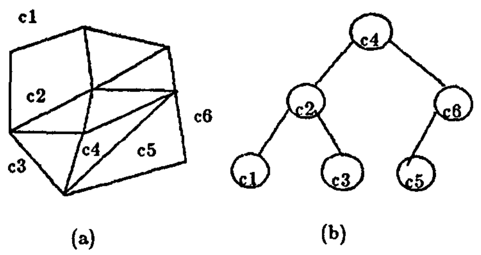

Figure 5: Decomposition of a planar subdivision into monotone chains.

Fact 10: [9] An N vertex (O(N) edges) PSLG can be decomposed into a set of monotone chains in O(NlogN) time and O(N) space. Alternatively, the procedure given in Cole and Sharir[3] can also be used for ordering the edges.

Algorithm VISIBLE

(1) We project the terrain map onto the X-Y plane and decompose the resultant planar graph into a set of monotone chains (with respect to the Y axis). The chains are ordered with respect to the X axis and gives us an ordering of "painting" the surface from front to back.

Comments : During any subsequent stage of the algorithm, we maintain an upper profile (of the y-z projection) of the part of the surface *processed that far. Notice that any point below (in the z direction) this upper profile of edges would not be visible if it belongs to a surface beyond the part of the image processed until then.*

(2) Pick up an edge from the monotone chain (obtained in stage 1) which is current chain being processed and find the intersections of this segment with the upper profile (which is again a monotone chain). The basic underlying problem is to detect the intersections of a line segment with a monotone chain quickly and accordingly update the upper profile. For this, we use the scheme described in the previous section.

(3) Repeat step 2 until no more edges are left.
Note that the actual displayed image is not the upper-profile itself; rather it is a PSLG with the faces (in the PSLG) belonging to a specific surface. This is updated as we detect intersections of segments with the current upper-profile. As soon as a visible region is detected, we can traverse its boundary (the bounding edges or vertices) from the ordered list of vertices of the upper profile and charge the cost to the visible face (region).

Lemma 3: At any time during the algorithm the space that can be used is at most $O(n\alpha(n)\log n)$ where $\alpha(n)$ is the inverse Ackermann's function.

Proof : This follows directly from the bound on the size of profile which is $O(n\ \alpha(n))$ for n line segments (Cole and Sharir[3]). This does not include the space for the displayed image which is O(k).

Lemma 4: Algorithm **Visible** runs in time $O((k + n)\log^2 n)$ and space $O(n\alpha(n)\log n + k)$ where n is related to the input size and k is the output size i.e. the number of edges and vertices in the planar graph representing the output image.

proof: Stage 1 of the algorithm requires O(nlogn) time (from fact 9). Stage 2 of the algorithm is dominated by the time to update O(log n) visibility polygons for insertion of a new visible subsegment. From Lemma 1, the total time for k (output size) intersections is $O(k \log^2 n)$. From Claim 1, Claim 2 and fact 6, we need the same time to update visibility polygons corresponding to the insertion of k segments. Compared to this, the total rebalancing cost is only O(klogn) from lemma 2. The bound for space follows directly from Lemma 3. □

Chazelle and Guibas[1] improve on the running time of their algorithm from the bounds stated in Lemma 1, by observing that their data structure can be made linear and then by a direct application of **fractional cascading**, which improve their running time by a factor of logn. The reason for this being the ability to search the visibility polygons in various levels given its position at some level in constant additional time. The cost for augmenting the datastructure to facilitate fractional cascading can be absorbed in the preprocessing cost. Our problem is more formidable since we have a dynamic environment - however the following result meets our requirements.

Fact 11[2] : Queries can be supported in time O(lognloglogn) and insertions and deletions can be in O(lognloglogn) in a dynamic data structure where a given item needs to be simultaneously located in O(logn) levels. The bounds for query is worst case, whereas they are amortized for insertions and deletions.

It can be shown that a single rebalancing operation of the dynamic data structure can be carried out in time proportional to O(th(v)loglogn). Furthermore, for a sequence of m operations or deletions we shall use the following result :
Fact 12 [2]: The rebalancing operations of the underlying $BB(\alpha)$ tree for a sequence of m insertions or deletions has cost O(mlognloglogn).
Facts 11 and 12 can be applied directly to lemma 4 leading to our main result of this section :

Theorem 1: There exists an algorithm for hidden surface elimination for terrain maps which runs in time O((k+n)lognloglogn) and space $O(n\alpha(n) + k)$ where n is the input size and k is the size of the displayed image.

IV A parallel algorithm for hidden surface elimination

Overview

The parallel algorithm is not a direct parallelization of the sequential algorithm developed in the previous sections but retains some of the key ideas. A major stumbling block is the sequential nature of detecting the polygon intersections with a line segment. In addition, because of our strong commitment to developing algorithms that are output sensitive, we use a model of parallel computation that is slightly different from the conventional PRAM model. Our main objective for an efficient parallel algorithm is to restrict the parallel running time to within polylogarithmic bounds and simultaneously keep the P*T (processor-time) product proportional to the output size of the displayed image. Since the final output size cannot be predetermined we assume a "pool" of free processors from where we can request processors as the algorithm progresses. The maximum number of processors busy at any instance of the algorithm is defined as the total number of processors needed by the algorithm. In the description that follows we describe the algorithm top-down followed by its analysis.

The main steps are :

1. Given a 2-D surface as a straight line graph in three dimensions, we project the line segments on the X-Y plane (the viewing direction is the negative x axis and the surface is a function $z = f(x,y)$). Because of the nature of terrain maps, no two projected segments will intersect.

2. If the graph is not triangulated, we triangulate the graph using Atallah, Cole and Goodrich[16] parallel triangulation. Since it is a planar graph, the number of edges and faces is still O(n) and from here our analysis will be in reference to the triangulated surface.

3. The triangulated graph is divided into two parts by a chain of edges monotonic to the y-axis by a method described later. In fact the triangulation is necessary to divide the image quickly and efficiently (in polylogarithmic time using only a linear number of processors). This process is repeated recursively on each of the halves until we have a constant number of edges in each group. Thus the depth of recursion can be at most O(logn). We will use a logn bit identifier for each trivial block where each bit represents the position with respect to a particular level (the MSB being the top level).

4. We now divide the graph into O(logn) stages in the following manner. In the first stage we divide the edges into two groups - one in which the MSB of the identifier is 1 and one which is 0. In the second stage we divide into four groups according to the first 2 bits being 00, 01, 10, and 11. In general in the kth stage ($k < \log n$) the edges are divided into 2^k groups according to the first k bits of the identifier.

```
5. For each level in parallel do
      for each group in parallel do
         Construct the profile of the edges. By profile we mean a
         function g which is the maximum (in z coordinate) of
         the projection of edges in the Y-Z plane. We will
         describe the method in detail shortly.

6. For each profile in parallel do
      Build the data structure of Chazelle and Guibas[1] to detect
      line segment intersection with the profile (which is a mono-
      tone polygon). We shall see that this data structure can be
      constructed quickly and efficiently.
```

7. We are now ready to compute the visibility of each segment in parallel. Because of the construction of the data-structure in step 7, we can allocate a processor to each segment and compute the intersections sequentially with the profiles in front of the segment. Note that there can be at most O(logn) profiles which determine the visibility of the segment. However, we cannot afford to compute all the intersections sequentially because

(i) The number of intersections can be large say n^ϵ $(0 < \epsilon < 1)$ and/or

(ii) Some of the intersections computed may not be visible in the final displayed image which will make the P*T product much larger than the output size.

To tackle the first problem, we can do a divide and conquer on the length of the segment. We start from a point in the middle of the segment (by middle we mean that the number of segments of the profile is nearly equal on both sides). We detect the first intersection point on the left and right size. If we find an intersection say on the right side we divide the remaining interval on the right into equal parts, request for an extra processor from the "pool" of processors and repeat the procedure. Notice that the maximum depth of recursion can be at most O(log k) where k is the number of intersections.

The problem posed by ii makes the previous approach inadequate. To ensure that we do not compute too many redundant intersections, we have to find a way to compute the visibility of a segment with respect to the actual profile. Computing all the n profiles in parallel may lead to a high degree of redundancy unless we are careful not to compute the same visible portions of the image repeatedly at various profiles. For this we need a parallel data structure to share the common portions of the profiles to keep the total number of computations minimal (comparable to the sequential case).

In the following paragraphs we describe each individual step in more detail and also analyze the running time of the algorithm. We do not address the issue of processor allocation in the analysis. Using a randomized algorithm given by Miller and Reif, the time bounds increase by a factor of O(logn) without affecting the processor bounds.

Lemma 5: Given n non-intersecting segments we can partition the set of edges into two sets S_1 and S_2 such that no edge of S_2 occludes any edge in S_1. Moreover the sets S_1 and S_2 are nearly equal (each of them is atmost twice the size of the other) and this can be done in $O(\log^3 n)$ time using n/logn processors in a PRAM model.

Proof: Consider a median line separating the end-points of the segments into two equal size sets. The median line is vertical to the horizontal plane containing the projection of the segments (and is also in the viewing direction). This line will intersect some of the segments which can be totally ordered with respect to the viewing direction from "back" to "front". Call the vertex induced graph on the left as G_l, the one on the right G_r and the set of line segments intersecting as G_m. Notice that G_l and G_r have no vertices in common and can be ordered independently of each other. Denote the segments in G_m as s_i and its end-points as l_i (left end-point) and r_i (right end-point). Recall that the graph is triangulated. Assume inductively that G_l and G_r have been ordered such that there are monotonic chains separating the the two graphs. Moreover, there is a tree of such "separating-chains" for either graphs. For each vertex l_i and r_i, we keep track of the chains it belongs to (a vertex may be part of more than one chain). Let max(l,i) and min(l,i) denote the maximum and the minimum of the part of the G_l in front of l_i. Analogously, define max(r,i) and min(r,i) for r_i. It follows that any separating-chain containing s_i can divide the graph into a ratio β where β is determined by max(l,i), min(l,i), max(r,i), min(r,i) and i. For example using the max on left and min on right the graph can be partitioned into a ratio $\frac{(\max(l,i)+(\min(r,i)+i)}{(\,|\,G_l-\max(l,i)\,|\,)+(\,|\,G_r-\min(r,i)\,|\,)+k-i}$ where $|G_m| = k$. We claim that there exists a segment in G_m that separates the graph into the required ratio $\beta \in [1/3, 2/3]$. Figure 6 shows an example of a separating chain. Suppose not, then there are two

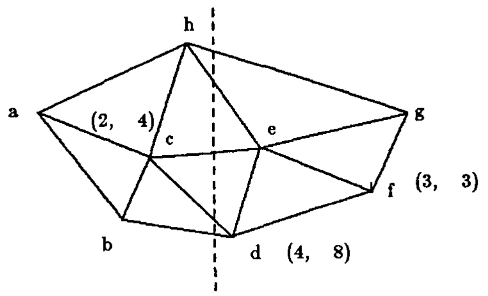

Figure 6: acef is a separating chain. The figures in parenthesis indicate the minimum and maximum number of edges in a chain ending at that vertex.

segments s_l and s_{l+1} such that the ratio is less than 1/3 for s_l and greater than 2/3 for s_{l+1}. Wlog, assume that in one case the front part consists of less than 1/3 of the edges and then there is a jump of 1/3. Notice that two consecutive edges in G_m have one vertex in common (since the graph is triangulated) so while moving from s_l to s_{l+1}, one of the endpoints (wlog the left endpoint) is fixed which also fixes the left separating-chain. Thus on the right graph G_r, the number of edges between the chains CR_l and CR_{l+1} is more than n/3. Since from the inductive hypothesis, we have a tree of separating chains for G_r, there exists a separating chain C_r which when concatenated with the separating chain on the left, divides the graph in the required ratio. For the inductive hypothesis to remain invariant we repeat the process recursively for the graph in front and back (for both sides of the separating chain). Thus we get a recurrence relation of the form:

$$T(n) \leq T(2n/3) + O(\log^2 n) \quad \text{yielding } T(n) = O(\log^3 n).$$

The processor requirement is obviously linear in the input size. But we can slow down the initial sorting (for finding median) by a factor of O(logn) since we need $O(\log^2 n)$ at each level and hence cut down on the processor requirements by O(logn) □

Corollary 2 : The set of segments can be partitioned into constant size groups of segments with the property that the groups are totally ordered in the direction from the viewing plane in $O(\log^4 n)$ time using $O(\frac{n}{\log n})$ processors.

This is achieved by using the above procedure recursively. Moreover, there are partitions of groups of powers of 2 i.e. partitions for groups of sizes $O(\frac{n}{2^i})$

Lemma 6 The profile of a group of k segments can be constructed in $O(\log^2 k)$ time using $O(\alpha(k)\frac{k}{\log k})$ processors in a PRAM model.

Proof: This is done by dividing the segments into two equal parts (arbitrarily), computing the profiles recursively and then merging the profiles as follows. Since we know that the profile of k/2 segments can have size at most $k\alpha(k)$ we can merge the endpoints of segments constituting the profiles in O(logk) time using $O(\alpha(k)k/\log k)$ processors. Find the predecessor of a point in the other set (which can be simultaneously computed while merging). From this we can determine if the point is visible (i.e. if it is a part of the resultant profile) by checking if it lies below a segment (one of whose end-points is the predecessor). We can now determine the intersections of the profiles using this information in the following manner. Note that every intersection involves at least one segment with the property that exactly one of its end-points is visible (see Figure 7).

All that needs to be done is finding the intersection which can again be determined from the predecessor information. The total time bound follows from the recursive application of this procedure. □

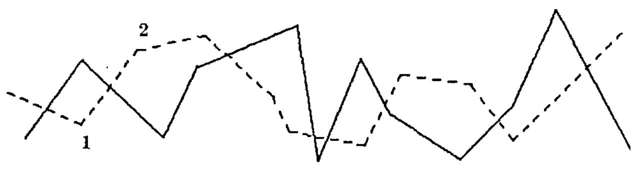

Figure 7: Merging two profiles in parallel.

Lemma 7 : The profiles of all the partitions (of sizes $\frac{n}{2^i}$ can be computed in $O(\alpha(n)\log^2 n)$ time (or $\alpha(n)\log^3 n$) using a linear $(O(\frac{n}{\log n}))$ number of processors in a CREW PRAM model.

Proof: Observe that each segment can be involved in at most O(logn) groups (for each value of i in $\frac{n}{2^i}$). The proof follows from lemma 10 and an application of slow down.

Lemma 8: The intersection(s) of the boundaries of two convex polygons with L and M vertices respectively can be computed optimally in O(log(L+M)) time using (L+M)/log(L+M) processors.

Proof: Follows from a straightforward parallelization of an algorithm given in Shamos & Hoey[76] and an optimal merging algorithm given by Shiloach and Vishkin[81].

Fact 13[Atallah, Cole & Goodrich[16]) : The data structure for fractional cascading on a given graph G of size n can be constructed optimally in O(logn) time using n/logn processors in a CREW PRAM model.

Lemma 9: For a profile of size P, we can construct the data structure of Chazelle and Guibas[1] for detecting intersections with a line segment in $O(\log^2 P)$ time using P/logP processors.

Proof: A profile (which is monotonic) can be divided recursively into halves (quarters etc) by sorting. We can sort P vertices in $O(\log^2 P)$ time using P/logP processors. Building the visibility tree will require O(logP) time for each level of the tree (proceeding from the leaves towards the root) from lemma 13. From fact 13, the data structure for fractional cascading can be constructed within the same time bounds.

Lemma 10: Given a profile P and a line segment s, we can find all the k intersections of the line segment with the profile in $O(\log n + \log^2 k)$ time using O(k/logk) processors in a CREW PRAM model.

Proof: We first find the diagonal of the profile such that the segment covers roughly equal number of diagonals on either side. This can be done by a simple binary search of the endpoints of the segment. Then we divide the line segment into two rays (in opposite directions) and do the sequential algorithm. If there is an intersection we allocate an extra processor to the part of the segment between the original endpoint and repeat the above procedure recursively. Clearly all the intersections will be detected within O(logn) recursive calls and the total number of processors required is at most $1 + 2 + 4 + .. k < 2k$. By slowing down, (i.e. finding O(logk) intersections sequentially by each processor) the result follows. □

For the rest of the algorithm consider a binary tree whose leaves correspond to the actual profiles at the groups with constant number of edges. Any internal node corresponds to the profile of the edges corresponding to the leaves of the sub-tree of which it is the root. To compute the actual profiles at each of the leaves, we take an approach similar to the parallel prefix computation. Starting from the root of the tree the computation proceeds towards the leaves level by level and so after O(logn) stages we have all the profiles and the data-structure for detecting intersections. A crucial factor is sharing of common visible segments between nodes in the same level. For example, a visible portion may be a part of the profiles in the first, second and third group in the partition of segments into groups of size of n/4 segments. This "repetition" may multiply at lower levels leading to a very inefficient algorithm since we have to build the data-structure repeatedly on the same parts of the profile again and again. Thus the total number of computations during the course of the algorithm may turn out to be several times larger than the output size, thus jeopardizing our initial objective of designing output size sensitive algorithms. Though there is some repetition i.e. redundant computation but it is within a factor of O(logn) as we shall show later. Our main procedure merges two profiles such that the cost of merging is no more than a few logarithmic factors from the output size. Before we shall analyze it more rigorously, we need to review some known results.

Fact 14: [Mehlhorn81] The class of BB(α) trees supports the full repertoire of concatenable queues. More specifically, two trees in the class of BB(α) trees can be concatenated in time proportional to $O(\log |S_1|/|S_2|)$ where S_1 and S_2 are the size of the trees. The operation SPLIT can also be implemented in O(log |S|) time where |S| is the size of the tree.

Fact 15:[Mehlhorn84] There exists a constant c, such that a node in a BB(α) tree does not go out of balance (i.e. there is no need for rotations or double rotations) before cL transactions (insertions or deletions) pass through that node where L is the number of leaves in the subtree of which it (the node) is the root.

Lemma 11: The number of rotations required to concatenate two BB(α) trees geometrically decreases towards the root (higher up in the tree).

Proof: Follows from Fact 15 and the procedure for concatenation outlined in Mehlhorn[81].

Lemma 12: Two profiles of size N_1 and N_2 can be merged in time proportional to $O(\log^2 k + \log^2 N)$ where $N = \min\{N_1, N_2\}$ using O(k/logk + N/logN)) processors where k is the number of intersections between the two profiles.

Proof (sketch): Assuming that we already have the data structure for finding intersections with line segments, we use the data structure corresponding to the larger profile and find the intersections of the segments constituting the smaller profile with the larger profile. From lemma 10, this can be done in $O(\log (N_1+N_2)+\log^2 k)$ time using O(k/logk) processors. We now have to update the data structure (for the merged profiles). For each new chain of segments consisting of a constant number of edges, we can use a procedure similar to the sequential case and hence update at a cost of $O(\log^2(N_1+N_2))$ per intersection. For a chain of edges longer than a constant, we use the data structure on this chain of edges and then merge this by using the operations SPLIT and CONCATENATE which can be carried out at roughly logarithmic cost from fact 14. The rebalancing operations can be more expensive, but using ammortized analysis we can bound the cost by charging the rebalancing operations to the newly inserted segments. From Claim 3, and the fact that a rebalancing operation for a node is proportional to the thickness of the node (as in the sequential case) we need to charge only a constant number of rebalancing operations to each segment inserted. Each rebalancing operation involves intersection of convex polygons which can be done in O(logN) time (lemma 8) using an optimal number of processors. In the case of parallel algorithm we shall charge the number of processors used (instead of the total number of operations) to the new segments inserted. The total time is the sum of the time used to insert an individual segment and the rebalancing operations which is bounded by $O(\log^2(N_1+N_2))$. The total number of processors used is clearly O(k/logk + N/logN). □

Lemma 13: The above procedure for merging two profiles runs in time $O(\log^3 N)$ for each level of the profile computation tree using $O((n+k)/\log(n+k))$ processors (which is sensitive to the output size at each level of the profile-computation tree).

Proof (sketch): As we go down the profile computation tree we need to keep track of the shared visible portions of the image but because of rotations the data-structure corresponding to the same portion may be different for different nodes of the profile-computation tree. This gives rise to the need for a pointer tree by which we can distinguish between the different structures of the same visible portion. The depth of this can be at most $O(\log n)$ and hence the claim follows. (Instead of a pointer to its child a node has a pointer tree and depending on which node of the profile-tree we are we choose the corresponding child).

Theorem 2 : There exists a parallel algorithm for hidden surface elimination on terrain maps that runs in time $O(\log^4 n)$ using $O(\frac{n+k}{\log(n+k)})$ processors and $(n+k)\log n$ space in a CREW PRAM model where n and k are the input and output sizes respectively.

Proof: For each level of the computation tree the algorithm requires $O(\log^3 n)$ time and hence the time bound follows the previous lemma and corollary 2. The processor bound follows from lemma 12. The space bound follows from the fact that there is a redundancy of a factor of at most logn because of the depth of the profile computation tree. □

V. Concluding remarks

In the previous sections, we presented sequential and parallel algorithms for hidden surface elimination for terrain maps. The running time of the sequential algorithm is proportional to the size of the output image, thus achieving the basic objective of this paper. However, the performance of our algorithm is not optimal in the worst case; in fact it is not clear what is the optimal running time for such a class of algorithms (which depends on the output size). It is suboptimal by a factor of lognloglogn in the worst case (which is achieved by an algorithm in [7]). Since the hidden surface algorithm is harder than the intersection problem (reporting all the k intersections of n line segments) we conjecture that a running time of $O((k+n)\log n)$ will be hard to improve upon.

The parallel algorithm is one of the first to be presented for this problem and though it may not be very practical in its present form, it sheds light on a class of parallel algorithms which are also output sensitive (in the processor bounds). We do not claim that the bounds provided in this paper are tight; to the contrary we feel that these may be considerably improved. Note that, compared to the sequential algorithm the parallel algorithm is less efficient by a factor of $O(\frac{\log^2 n}{\log\log n})$.

Also, our computations are based on the assumption that the viewer is located at infinity i.e. the projections are orthonormal. A more realistic image can be obtained placing the viewer at a finite distance (and hence obtaining a perspective view of the terrain) and modifying the algorithm suitably though it is not obvious that such a modification can be done easily.

A natural direction for further work is to generalize the algorithm for hidden-surface elimination for any surface. For that we need efficient algorithms for ordering the edges quickly and also generalize the intersection detection algorithm of Chazelle & Guibas[1] to polygonal boundaries that may have 'holes' inside.

References

[1] Chazelle B. and Guibas L.N., "Visibility and Intersection Problems in Plane geometry," Proc. ACM Symp.on Computational geometry, 1985, pp. 135-146.

[2] Mehlhorn K. and Naher S., "Dynamic Fractional cascading,".

[3] Cole R. and Sharir M., Visibility problems for polyhedral terrains," Tech. Rept. No. 92, Courant Institute of Math. Sc., Dec, 1986.

[4] Mehlhorn K : "Data Structures and Algorithms," Springer Publ. Comp, 1984
a) Vol. 1: Sorting and Searching
b) Vol. 3: Multidimensional Searching and Computational Geometry.

[5] Preparata F. and Shamos I., "Computational Geometry: an introduction," Springer Publ, 1985.

[6] Chazelle B., "A Theorem on Polygon Cutting with Applications," Proc. of IEEE FOCS 1982, pp. 339-349.

[7] McKenna M., "Worst-case optimal hidden-surface removal," Tech. Rept JHU/EECS-86/05.

[8] Wright T.J., "A Two-space solution to the hidden line problem for plotting functions of two variables," IEEE Trans. on Comput., vol. c-32, no. 1, pp. 28-33.

[9] Lee D.T. and Preparata F.P., "Location of a point in a planar subdivision and its applications, SIAM Journal of Comput., 6(3), 1977, pp. 594-606.

[10] Preparata F.P., "An optimal real time algorithm for planar convex hulls," Comm. of the ACM, 22, 1979, pp. 402-405.

[11] Chazelle B. and Dobkin D., "Intersection of Convex Objects in Two and Three Dimensions," JACM, vol 34, Jan 1987, pp 1-27.

[12] Mehlhorn K., "Arbitrary weight changes in Dynamic trees," Theoretical Informatics, vol 15, 1981, pp 183-211.

[13] Nurmi O., "A fast line-sweep algorithm for hidden line elimination," BIT, vol 25, 1985, pp 466-472.

[14] Schmitt A., "Time and Space bounds for hidden line and hidden surface elimination algorithms," EUROGRAPHICS '81, pp 43-56.

[15] Goodrich M.T., "A polygonal approach to hidden-line elimination," Tech Rept 87-18, Dept of Computer Science, Johns Hopkins University.

[16] Atallah M.J., Cole R. & Goodrich M.T., "Cascading Divide-and-Conquer: A technique for Designing Parallel Algorithms," Proc. of the 28th Annual Symp. on FOCS, 1987, pp. 151- 160.

[17] Sutherland I.E., Sproull R.F. & Schumacker R.A., "A Characterization of Ten Hidden-Surface Algorithms," Computing Surveys, vol 6, no.1, March 1974, pp. 1-25.

Appendix 1

BB(α) trees are a class of weight-balanced trees i.e the number of nodes in the subtrees are balanced. If T is a binary tree with left subtree T_l and right subtree T_r and α a fixed real in the range [1/4, 1- $\sqrt{2}/2$], then T is of bounded balance α if for every subtree $\overline{T}$ of T
$\alpha \leq \rho(\overline{T}) \leq 1 - \alpha$ where $\rho(T) = | T_l | = 1 - |T_r|/|T|$.
BB(α) trees have the following properties:
(i) they have logarithmic depth: Height(T) $\leq 1 + (\log(n+1) - 1)/\log(1/1 - \alpha))$
(ii) they have logarithmic average path length

Lemma 3 [Mehlhorn] : For all $\alpha \in (1/4, 1 - \sqrt{2}/2]$ there are constants $d \in [\alpha, 1 - \alpha]$ and $\delta \geq 0$ such that for T, a binary tree with subtrees T_l and T_r and
(1) T_l and T_r are in BB(α)
(2) $|T_l|/|T| < \alpha$ and either
2.1 $|T_l|/(|T| - 1) \geq \alpha$ implying that an insertion into T_r occurred or
2.2 $(|T_l|+1)/(|T|+ 1) \geq \alpha$ implying that a deletion from left subtree had taken place.
2.3 ρ_2 is the root balance of T_r

then (i) if $\rho_2 \leq d$ then a rotation rebalances the tree
(ii) if $\rho_2 > d$ then a double rotation rebalances the tree.

Figure 4 shows the rotation and double rotation operations and the corresponding changes in the root balances. Note that d and δ are functions of α.

Lemma 4: [Mehlhorn] There is a constant c such that the total number of rotations and double rotations required to process an arbitrary sequence of m insertions and deletions into an initially empty BB(α) tree is $<$ cm and the total number of rebalancing operations over all the vertices of level i $BO_i(v) = O(m\ (1-\alpha)^i)$. (Note that i increases as we go up the tree which implies that rebalancing operations are very rare as we get closer to the root.)